Levels of Abstraction in Operating Systems

Robert L. Brown
Peter J. Denning
Walter F. Tichy

July, 1984

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS TR 84.5

RIACS

Research Institute for Advanced Computer Science

Levels of Abstraction in Operating Systems

Robert L. Brown
Peter J. Denning
Walter F. Tichy

Abstract:

By defining the user's view of a computer system and aiding in every command, the operating system plays a critical role in computing. Operating systems can be modeled by a hierarchy of levels corresponding to important time scales or object sizes within the computer system.

Key words and Phrases:

Abstract data types, distributed file system, distributed system, hierarchical design, levels of abstraction, network operating system, object-oriented system, operating system.

Work on this article was supported by the National Aeronautics and Space Administration under Contract NAS2-11530

Authors' addresses: Robert L. Brown, RIACS, NASA Ames Research Center, MS 230-5, Moffett Field, CA 94035 (net address: brown@riacs); Peter J. Denning, RIACS, NASA Ames Research Center, MS 230-5, Moffett Field, CA 94035 (net address: denning@riacs); Walter F. Tichy, Computer Sciences Department, Purdue University, West Lafayette, IN 47907 (net address: tichy@purdue).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

The operating systems of 1955 were control programs a few thousand bytes long that scheduled jobs, drove peripheral devices, and billed users. The operating systems of 1984 are much larger both in size and in responsibility. The largest ones, such as Honeywell's Multics or IBM's MVS, are tens of millions of bytes long. Intermediate ones, such as Bell Labs's UNIX or Digital Equipment's VMS, are several hundreds of thousands of bytes long. Even the smallest, most pared-down systems for personal computers are tens of thousands of bytes long.

The intellectual content of the field of operating systems was recognized in the early 1970s. Virtually every curriculum in computer science and engineering includes a course on operating systems. Texts are numerous. The continuing debates -- over the set of concepts that should be taught and over the proper mix between concepts and implementation projects -- are signs of the vitality of the field.

Since 1975, personal computers for home and business have grown into a multi- billion dollar industry. Advanced graphics workstations and microcomputers have been proliferating. Local networks -- e.g., Ethernet, ring nets, wideband nets -- and network protocols -- e.g., X.25, PUP, TCP/IP -- allow large systems to be constructed from many small ones. The available hardware has grown rapidly in power and sophistication.

In view of the rapid advances in power and sophistication of available hardware, it is natural to ask: Will hardware eventually obviate software control programs? Is the intellectual core recorded in operating systems texts outmoded? Is operating systems a dying field? In this paper we will argue that the power and complexity of the new hardware intensifies the need for operating systems, that the intellectual core contains the concepts needed for today's computer systems, and that operating systems are essential.

1.1. Definitions of Operating System

Before looking into these questions, we need to agree on a definition of "operating system". The oldest definition, which says *an operating system is a control program for allocating resources among competing tasks* describes only a small portion of a modern operating system's responsibilities. This definition is inadequate.

Among the great problems faced by operating systems designers is managing the complexity of operations at many levels of detail, ranging from hardware operations that take one billionth of a second to software operations that take tens of seconds. An early strategem was *information-hiding* -- confining the details of managing a class of "objects" within a module that has a good interface with its users. With information-hiding, the designers can protect themselves from extensive reprogramming if the hardware or some part of the software changes: the change affects only the small portion of the software interfacing directly with that system component. This principle has been extended from isolated subsystems to an entire operating system. The basic idea is to create a hierarchy of levels of abstraction, so that at any level one can ignore the details of what is going on at all lower levels. At the highest level is the user of the system, who ideally is insulated from everything except what he aims to accomplish. As a consequence of these developments, a better definition today is, *an operating system is a set of software extensions of primitive hardware, culminating in a virtual machine that serves as a high level programming environment.*

Operating systems of this type can support diverse environments: programming, text processing, real-time processing, office automation, database, and hobbyist.

1.2. Current Operating Systems

Most operating systems for large mainframes are direct descendants of third generation systems -- e.g., Honeywell Multics, IBM MVS and VM/370, and CDC Scope. These systems introduced important concepts such as timesharing, multiprogramming, virtual memory, sequential processes cooperating via semaphores, hierarchical file systems, and device independent I/O [Denn71, Denn76].

During the 1960s, there were many projects to construct timesharing systems and test the many new operating systems concepts. These included MIT's Compatible Time Sharing System (CTSS), the University of Manchester Atlas, the University of Cambridge Multiple Access System (CMAS), IBM TSS/360, and RCA Spectra/70. The most ambitious project of all was Multics (Multiplexed Information and Computing Service) for the General Electrical 645 (later renamed Honeywell 6180) processor [Orga72]. Multics simultaneously tested new concepts of processes, interprocess communication, segmented virtual memory, page replacement, linking new segments to a computation on demand, automatic multiprogrammed load control, access control and protection, hierarchical file system, device independence, I/O redirection, and a high-level language shell.

Another important concept of third generation systems was the virtual machine, a simulated copy of the host. Virtual machines were first tested around 1966 on the M44/44X project at the IBM T. J. Watson Research Center. In the early 1970s virtual machines were used in IBM's CP-67 system, a time sharing system that assigned each user's process to its own virtual copy of the IBM 360/67 machine. This system has been moved to the IBM 370 machine and is now called VM/370 [Gold74, IBM73]. Because each virtual machine can run a different copy of the operating system, VM/370 is effective for developing new operating systems within the current operating system. But because virtual machines are well isolated,

communication among them is expensive and awkward.

Perhaps the most influential current operating system is UNIX, a complete reengineering of Multics for the DEC PDP family of computers. It is an order of magnitude smaller than Multics. It retains the most useful concepts of Multics -- processes, hierarchical file system, device independence, I/O redirection, and a high-level language shell. It dispensed with virtual memory and the detailed protection system; it introduced the pipe. It offered a large library of utility programs that were well integrated with the command language. Most of UNIX is written in a high-level language, C, which has allowed it to be transported to a wide variety of processors, from mainframes to personal computers [Ritc74, Kern84].

In systems consisting of multiple UNIX machines connected by a high-speed local network, it is desirable to hide the locations of files, users, and devices from those who do not wish to deal with those details. LOCUS is a distributed version of UNIX that accomplishes this by means of a directory hierarchy that spans the entire network [Pope81].

In recent years a large family of operating systems has been developed for personal computers. These include MS-DOS, PC-DOS, APPLE-DOS, CP/M, Coherent, and Xenix. These are all simple systems with limited function, designed for 8- and 16-bit microprocessor chips with small memories. In many respects, the development of personal computers is repeating the history of mainframes in the early 1960s -- for example, multiprocess operating systems for microcomputers have appeared only recently in the forms of pared-down UNIX-like systems such as Coherent and Xenix. Because only the large firms can sell enough machines to make their own operating systems viable, there is strong pressure for standard operating systems. The emerging standards are PC-DOS, CP/M, and UNIX.

Research on operating systems continues. There are numerous experimental systems exploring new concepts of system structure and distributed computation. The operating

system for the Cambridge CAP machine exploits the hardware's microcode support for capability addressing to implement a large number of processes in separately protected domains. Data abstraction is easy to implement on this machine [Wilk79].

StarOS is an operating system for the Cm* machine. Its central purpose is the supportof the "task force," a group of concurrent processes cooperating in a computation. StarOS also uses capabilities to control access to objects [Jone79]. Another operating system for the Cm* machine is Medusa. It is composed of several "utilities", each implementing a particular abstraction such as a file system. Each utility can include several parallel processes running on separate processors. There is no central control [Oust80].

Grapevine is a distributed database and message delivery system used widely within the Xerox Corporation. The network contains special nameservers that can find the locations of users, groups, and other services when given their symbolic names. There is no central control and it can survive the failures of the nameserver machines [Birr82]. Because it does not provide all the services of a high-level programming environment, Grapevine is not a true operating system.

The "V kernel" is an experimental system aiming for efficient, uniform interfaces between system components. A complete copy of the kernel runs on each machine of the network and hides the locations of files, devices, and users. V is a descendent of THOTH, an earlier system worked on by the author of V [Cher84, Cher82].

The Provably Secure Operating System (PSOS) is a level-structured system whose high-level code has been proved correct in the context of a rigorous hierarchical design methodology developed at SRI International [Neum80]. Although it was intended for secure computing, PSOS explored many principles that can help any operating system toward the goal of provable correctness.

These examples demonstrate that the new technology has created new control problems for operating systems designers to solve. The need for operating systems is stronger than ever.

2. MODEL OF AN OPERATING SYSTEM

The hierarchical structure of a model operating system separates its functions according to their characteristic time scales and their levels of abstraction. Table 1 shows an organization spanning fifteen levels. It is not a model of any particular operating system but rather incorporates ideas from several systems. It includes facilities for distributed processing.

Each level is the manager of a set of "objects", which can be hardware or software and whose nature varies greatly from level to level. Each level also defines operations that can be carried out on those objects. The levels obey two general rules:

1. *Hierarchy.* Each level adds new operations to the machine and hides selected operations at lower levels. The set of operations visible at a given level form the instruction set of an abstract machine. Hence a program written at a given level can invoke visible operations of lower levels but no operations of higher levels.
2. *Information Hiding.* The details of how an object of given type is represented or where it is stored are hidden within the level responsible for that type. Hence no part of an object can be changed except by applying an authorized operation to it.

The principle of data abstraction embodied in the levels model traces back to Dennis and Van Horn's 1966 paper, which emphasized a simple interface between users and the kernel [Denn66]. The first instance of a working operating system whose kernel spanned several levels was reported by Dijkstra in 1968 [Dijk68]. The idea has been extended to generate families of operating systems for related machines [Habe76] and to increase the portability of an operating

TABLE 1: An Operating System Design Hierarchy.

Level	Name	Objects	Example Operations
15	Shell	User programming environment scalar data, array data	statements in shell language
14	Directories	Directories	create, destroy, attach, detach, search, list
13	User Processes	User process	fork, quit, kill, suspend, resume
12	Stream I/O	Streams	open, close, read, write
11	Devices	External devices and peripherals such as printer, display, keyboard	create, destroy, open, close, read, write
10	File System	Files	create, destroy, open, close, read, write
9	Communications	Pipes	create, destroy, open, close, read, write
8	Capabilities	Capabilities	create, validate, attenuate
7	Virtual Memory	Segments	read, write, fetch
6	Local Secondary Store	Blocks of data, device channels	read, write, allocate, free
5	Primitive Processes	Primitive process, semaphores, ready list	suspend, resume, wait, signal
4	Interrupts	Fault handler programs	invoke, mask, unmask, retry
3	Procedures	Procedure segments, Call stack, display	mark_stack, call, return
2	Instruction Set	Evaluation stack, microprogram interpreter,	load, store, un_op, bin_op, branch, array_ref, etc.
1	Electronic Circuits	Registers, gates, busses, etc.	clear, transfer, complement, activate, etc.

system kernel [Cher82]. The Provably Secure Operating System (PSOS) is the first complete level-structured system reported and formally proved correct in the open literature [Neum80].

We now turn to a brief summary of each level in Table 1. Greater detail follows in later sections.

2.1. The Single-Machine Levels: 1-8

Levels 1-8 are called *single-machine levels* because their operations are well understood from primitive machines and require little modification for advanced operating systems.

The lowest levels include the hardware of the system. Level 1 is the electronic circuitry, where the objects are registers, gates, memory cells, and the like, and the operations are clearing registers, reading memory cells, and the like. Level 2 adds the processor's instruction set, which can deal with somewhat more abstract entities such as an evaluation stack and an array of memory locations. Level 3 adds the concept of a procedure and the operations of call and return Level 4 introduces interrupts and a mechanism for invoking special procedures when the processor receives an interrupt signal.

The first four levels correspond roughly to the basic machine as it is received from the manufacturer, although there are some interactions with the operating system. For example, interrupts are generated by hardware but the interrupt-handler routines are part of the operating system.

Level 5 adds primitive processes, which are simply single programs in the course of execution. The information required to specify a primitive process is its stateword, a data structure that can hold the values of the registers in the processor. This level provides a context switch operation, which transfers the attention of of a processor from one process to another by saving the stateword of the the first and loading the stateword of the second. This

level contains a scheduler that selects, from a "ready list" of available processes, the next process to run after the current process is switched off the processor. This level also provides semaphores, the special variables used to cause one process to stop and wait until another process has signalled the completion of a task. This level has a simple hardware implementation [Denn81]. Primitive processes are analogous to *system processes* in PSOS and *lightweight processes* in LOCUS.

Level 6 handles access to the secondary-storage devices of a particular machine. The programs at this level are responsible for operations such as positioning the head of a disk drive and transferring a block of data. Software at a higher level determines the address of the data on the disk and places a request for it in the device's queue of pending work; the requesting process then waits at a sempahore until the transfer has been completed.

Level 7 is a standard virtual memory, a scheme that gives the programmer the illusion of having a main memory space large enough to hold the program and all its data even if the available main memory is much smaller [Denn70]. Software at this level handles the interrupts generated by the hardware when a block of data is addressed when it is not present in the main memory; this software locates the missing block in the secondary store, frees space for it in the main store, and requests Level 6 to read in the missing block.

Level 8 implements capabilities, which are unique internal addresses for software objects definable at higher levels. This level allows capabilities to be read, but not altered. This level provides a validate operation that enables the programmer of higher-level procedures to verify that actual parameters are of the expected types.

Up through level 8, the operating system deals exclusively with the resources of a single machine. Beginning with the next level, the operating system encompasses a larger world including peripheral devices such as terminals and printers and also other computers attached

to the network. In this world, pipes, files, devices, user processes, and directories can be shared among all the machines.

2.2. The Multi-Machine Levels: 9-14

Every object in the system has two names: its "external name", a string of characters having some meaning to users, and its "internal name", a binary code used by the system to locate the object. The mapping from external to internal names is controlled by the user through directories. The mapping from internal names to physical locations is controlled by the operating system, giving it the ability to move objects among several machines without affecting any user's ability to use those objects. This principle, called *delayed binding*, was important in third generation operating and is even more important today [Denn71].

To hide the locations of all sharable objects, both external and internal names must be global, i.e., they can be interpreted on any machine. Unique external names can be constructed as pathnames in the directory hierarchy (defined at Level 14). Unique internal names are provided by capabilities (Level 8). If the local network communication system (Level 9) is efficient, software at the higher levels can obtain access to a remote object with little penalty.

Level 9 is explicitly concerned with communication between processes, which can be arranged through a single mechanism called a pipe. A pipe is a one-way channel: a stream of data flows into one end and out of the other. A request to read items is delayed until they are actually present in the pipe. A pipe can equally well connect two processes on the same machine or on different machines. A set of pipes linking levels in all the machines can serve as a broadcast facility, which is useful for finding resources that might be anywhere in the network [Bogg83]. Pipes are implemented in UNIX [Ritc74] and have been copied in recent systems such as iMAX [Orga83] or XINU [Come84].

Level 10 provides for long-term storage of named files. Whereas Level 6 deals with disk storage in terms of tracks and sectors -- the physical units of the hardware -- Level 10 deals with more abstract entities of variable length. Indeed a file may be scattered over many noncontiguous tracks and sectors. To be examined or updated, a file's contents must be copied between virtual memory and the secondary storage system. If a file is kept on a different machine, Level-10 software can create a pipe to Level 10 on the file's home machine.

Level 11 provides access to external input and output devices such as printers, plotters, and the keyboards and display screens of terminals. There is a standard interface with all these devices and again a pipe can be used to gain access to a device attached to another machine.

Level 12 provides a means by which user processes can be attached interchangeably to pipes, files, or devices for input and output. The idea is to make each of the fundamental operations of Levels 9, 10, and 11 (OPEN, CLOSE, READ, and WRITE) look the same so that the author of a program need not be concerned with the differences among these objects. This is achieved in two steps. First, the information contained in pipes, files, and devices is regarded simply as streams of bytes; requests for reading or writing move segments of data between streams and a user process. Second, a user process is programmed to request all input and output via *ports*, which are attached by the open operation at run time to specific pipes, files, or devices.

Level 13 implements user processes, which are virtual machines executing programs. It is important to distinguish the user process from the primitive process of Level 5. All the information required to define a primitive process can be expressed in the stateword that records the contents of the registers in the processor. A user process includes not only a primitive process, but also a virtual memory containing the program and its work space, a list

of arguments supplied as parameters when the process was started, a list of objects with which the process can communicate, and certain other information about the context in which the process operates. A user process is much more powerful than a primitive process.

Level 14 manages a hierarchy of directories that catalogue the hardware and software objects to which access must be controlled throughout the network: pipes, files, devices, user processes, and the directories themselves. The central concept of a directory is a table that matches external names of objects with capabilities containing their internal names. A hierarchy arises because a directory can include among its entries the names of subordinate directories. Level 14 ensures that the subhierarchies encached at each machine are consistent with one another.

The directory level is responsible only for recording the associations between the external names and capabilities; other levels manage the objects themselves. Thus when a directory of devices is searched for the string "laser", the result returned is merely a capability for the laser printer. The capability must be passed to a program at Level 11, which handles the actual transmission to that printer.

Level 15 is the "shell", so called because it is the level that separates the user from the rest of the operating system. The shell is the interpreter of a high level command language through which the user gives instructions to the system. The shell incorporates a listener program that responds to a terminal's keyboard; it parses each line of input to identify program names and parameters; it creates and invokes a user process for each program and connects it as needed to pipes, files, and devices.

2.3. General Comments on Level Structure

The level structure is a hierarchy of functional specifications. Its purpose is to impose a high degree of modularity and enable incremental verification, installation, and testing of the software.

In a functional hierarchy, a program at a given level may directly call any visible operation of a lower level. No information flows through any intermediate level. The level structure can be completely enforced by a compiler, which can insert procedure calls or expand functions in-line [Habe76]. A recent example of its use is in XINU, an operating system for a distributed system based on LSI 11/02 machines [Come84].

It is important to distinguish the level structure discussed here from the layer structure of the ISO (International Standards Organization) model of long-haul network protocols [Tane81]. In the ISO model, information is passed down through all the layers on the sending machine and back up through all the layers on the receiving machine. Each layer adds overhead to a data transmission, whether or not that overhead is required. Models for long-haul network protocol structure may not be efficient in a local network [Pope81].

A significant advantage of a functional levels over information-transferring layers is efficiency: a program that does not use a given function will experience no overhead from that function's presence in the system. For example, procedure calls will validate capabilities only when they are expected. Common objects (such as pipes, files, devices, directories, and user processes) are implemented by their own levels rather than as new "types" within a general type-extension scheme [Wulf81].

Each level should be able to locate local objects by their internal names without having to rely on a central mapping mechanism. This is not only a step toward reliability in a distributed network, but also efficiency because central mechanisms are prone to be

bottlenecks.

Operating system designers ought to take reasonable steps to verify that each level of the operating system meets its specifications. This too serves efficiency as well as reliability: run-time checks need be included in system procedures only for conditions that cannot easily be verified a priori. Thus, system procedure calls must check at run time that expected capabilities are present as parameters because the calling programs may be unverified; but other aspects of parameter type checking can be performed by a compiler.

In the following sections we will give more detail about the mechanisms from the capability level (Level 8) upwards.

3. DISTRIBUTED CAPABILITIES (Level 8)

The external names of sharable objects are character strings of arbitrary length having meaning to human users. Because these strings are difficult to manipulate efficiently, the operating system provides internal names for quick access to objects. One purpose of Level 8 is to provide a standard way of representing and interpreting internal names for objects.

To prevent a process from applying invalid operations to an object whose internal name it knows, the operating system can attach a type code and an access code to an internal name. The combination of codes (type, access, internal-name) is called a *capability*. All processes are prevented from altering capabilities. The system assumes that the very fact that a process holds a capability for an object is proof of its authorization to use that object; processes are responsible for controlling the capabilities they hold.

The simplest way to protect capabilities from being altered is to tag the memory words containing them with a special bit and to permit only one instruction, the "create-capability" instruction, to set that bit [Fabr74, Wilk79]. The IBM System 38 is a recent example of an

efficient system using tags to distinguish capabilities from other objects in memory [Levy84]. Capabilities were first proposed as an efficient method of implementing an object-oriented operating system [Denn66]. This has continued to be the main reason for using them [Wulf81, Levy84, Orga83].

All existing implementations of capabilities are based on a central mechanism for mapping the internal name to an object. These mappings are direct extensions of virtual memory addressing schemes [Denn76, Fabr74]. Unfortunately, a central mapping scheme cannot be used with a distributed system whose component machines may fail. So the responsibility for mapping must be distributed by allowing each the procedures of Levels 9-14 on each machine to read and interpret locally the fields within validated parameter capabilities.

The storage structure and mapping scheme for capabilities is illustrated in Figure 1. The name field of a capability of type *T* consists of a code *M* for the machine on which the capability was created and an index number *I*. The machine number is needed because some capabilities (those for open pipes, files, and devices) can only be used on the issuing machine. The access code specifies which of the *T*-type operations can be applied to the object. The index number *I* is used by the level in charge of *T*-type objects on machine *M* to address a descriptor block for the given object. The descriptor block records control information about an object, times and dates of creation and last update, and current size and attributes of the object. The location of the descriptor block denotes the location of the object - moving the descriptor block from one machine to another effectively moves the object.

The procedures implementing operations at Levels 9-15 must conform to certain standards that ensure the proper use of capabilities. One is an agreement on the codes for the object types; eight are listed in Table 2. We will use the notation *T_cap* to denote a

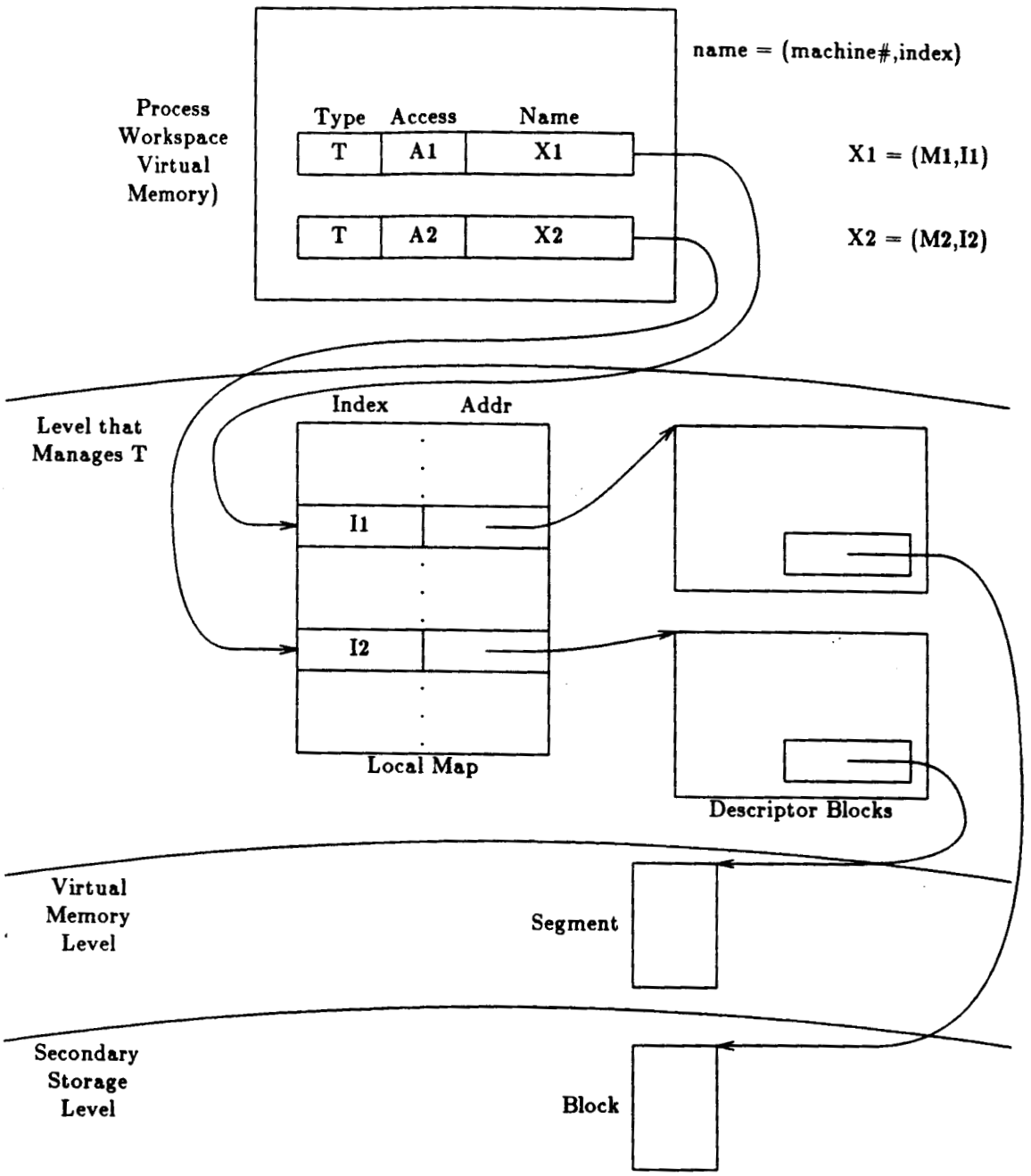

FIGURE 1. The storage structure for representing object consists of a chain starting with a capability, through a local map under the control of the object's level, through a descriptor block, to the object itself. A change in location of the object requires no change in any capability. The index numbers are generated locally by the level when it creates objects. In this example, the process holds two capabilities of type T; one of the objects is in a segment in virtual memory and the other is in a block of secondary storage.

capability of type T where T is one of the abbreviations in the table (e.g. file_cap).

The remaining standards concern the creation of capabilities pointing to new objects and the application of specific operations to those objects. Suppose Level L ($L > 8$) is the manager of T-type objects. This level contains a procedure to create new objects of type T and one or more procedures to apply given functions to objects of type T. The create operation must use a call of the form

T_cap := CREATE_T(initial-value)

This procedure performs all the steps required to set up the storage for a new object of type T: it obtains space in secondary storage for the object and stores in it the given initial value, it sets up a descriptor block, it finds an unused index and sets up the entry in the local map, and finally it creates a capability of type T (denoted "T_cap").

TABLE 2: Capability Type Marks and their Abbreviations.

Level	Type mark	Abbreviation
14	directory	dir
13	user process	up
11	device	dev
	open device	op_dev
10	file	file
	open file	op_file
9	pipe	pipe
	open pipe	op_pipe

Creating a capability is a critical operation. Level 8 implements a special operation for this purpose:

T_cap := CREATE_CAP(*I*)

where *I* is the local index number chosen by Level *L*. When used inside the CREATE_T operation on machine *M*, CREATE_CAP constructs a capability (*T*, *A*, *M*, *I*), sets to 1 the capability bit of the memory word containing it, and returns the result. The code for *M* comes from a register in the processor. The code for *A* is the one denoting maximum access. The code for *T* comes from a field in the program status word (PSW), a processor register that also contains the program counter of the current procedure. The compilers must be set up to generate type(PSW)=*T* only for the CREATE_T procedure and type(PSW)=null for all other procedures. CREATE_CAP fails if executed when type(PSW)=null.

The procedures for applying operations to a given object have the generic form

APPLY_OP(T_cap, parameters)

which means that OP(parameters) must be applied to the object denoted by T_cap. The compiler can validate that the first actual parameter on any call to APPLY_OP is indeed of type *T* by using another operation of Level 8, called VALIDATE, which checks that this parameter is a capability whose type code is *T* and whose access code enables operation OP. VALIDATE can also be used to verify the presence of other capabilities among the other parameters.

A procedure may reduce the access rights of a capability it passes to another procedure by using the Level-8 operation ATTENUATE.

Table 3 summarizes the operations implemented at Level 8.

4. COMMUNICATIONS (Level 9)

The communications level provides a single mechanism, the *pipe*, for moving information from a writer process to a reader process on the same or different machines. The most important property is that the reader must stop and wait until the writer has put enough data into the pipe to fill the request. Level 9 gives the higher levels the ability to move objects among the nodes of the network.

The external interface presented by the communications level consists of the commands in Table 4. When two communicating processes are on the same machine, a pipe between them can be stored in shared memory and the READ_PIPE and WRITE_PIPE operations are implemented the same as the *send* and *receive* operations for "message queues" [Brin73].

TABLE 3: Specification of Capability Operations (Level 8).

Form of call	Effect
T_cap := CREATE_CAP(*I*)	If the type-mark in the current PSW is non-null, create a new capability with type field set to that mark, access code maximum, machine field the local machine identifier, and index *I*.
VALIDATE(*p*, *n*, *(T1,a1)*,..., *(Tn,an)*)	Verify the capability at the caller's virtual address *p*. For at least one i=1,...,n the following must be true: the capability contains *Ti* in its type field and permits access *ai*. If *Ti* denotes op_pipe, op_file, or op_dev, the machine field must match the identifier of the local machine. (Fails if these conditions are not met.)
cap := ATTENUATE(cap, mask)	Returns a copy of the given capability with the access field replaced by the bitwise AND of *mask* and the access field from *cap*.

TABLE 4: Specification of Communication Level Interface (Level 9).

Form of call	Effect
pipe_cap := CREATE_PIPE( )	Creates a new empty pipe and returns a capability for it. (If the caller is a user process, it can store this capability in a directory entry and make the pipe available throughout the system.)
DESTROY_PIPE(pipe_cap)	Destroy the given pipe (undo a create pipe operation).
op_pipe_cap := OPEN_PIPE(pipe_cap, rw)	Opens the pipe named by the pipe capability by allocating storage and setting up a descriptor block. Initially, the pipe is empty. If rw=*write*, the open-pipe capability has its write permission set and can be used only by the process at the input end of the pipe. If rw=*read*, the open-pipe capability has its read permission set and can be used only by the process at the output end of the pipe. Does not return until both reader and writer have requested connections. (Fails if the pipe is already open for writing when rw=*write* or reading when rw=*read*.) If both sender and receiver are on the same machine, the open-pipe descriptor block will indicate that shared memory can be used for the pipe; otherwise a network protocol must be used.
READ_PIPE WRITE_PIPE CLOSE_PIPE	These have the same effects as the READ, WRITE, and CLOSE operations described in the section on stream I/O.
op_pipe_cap := BROADCAST(msg)	Broadcast a message to all type managers in the network that manage objects of the same type as the calling local type manager. Returns an open pipe capability for reading responses.

When the two processes are on different machines, the communications level must implement the network protocols required to move information reliably between machines. (See Figure 2.) These protocols are much simpler than long-haul protocols because congestion and routing control are not needed, packets cannot be received out of order, fewer error types are possible, and errors are less common [Pope81].

The read and write operations become ambiguous unless both a reader process and a writer process are connected to a pipe. Should a writer be blocked from entering information until the reader opens its end? What happens if either the reader or writer breaks its

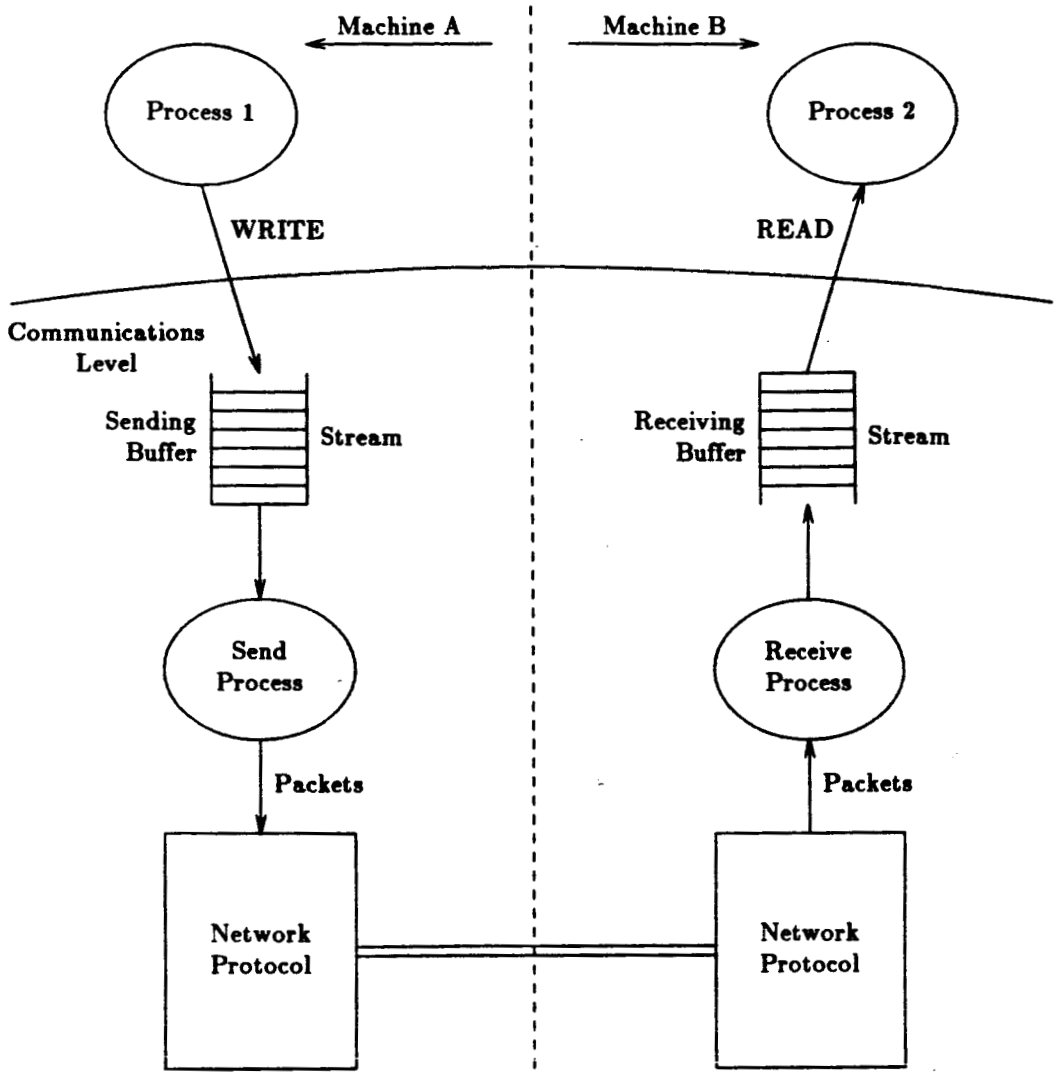

FIGURE 2. A network protocol must be used when two processes connected by a pipe are on different machines. The WRITE requests of the sender append segments to a stream awaiting transmission. The sender process transmits the stream as a sequence of packets, which are converted back into a stream and placed in the receiving buffer. Each READ request of the receiver waits until the requested amount of data is in the buffer then returns it.

connection? Questions like these are dealt with by a *connection protocol.* A simple connection protocol is called *rendezvous on open and close*:

1. The open-for-reading and the open-for-writing requests may be called at different times but both returns are simultaneous.

2. The close operation, executed by the reader, shuts both ends of the pipe; executed by the writer is deferred until the reader empties the pipe.

A pipe capability can be stored in a file or a message and passed to another machine over an existing open pipe or by broadcast. A pipe capability can also be listed in a directory (to be discussed later), making the pipe a global object. (In this case it is like a "FIFO file" in System-5 UNIX [Bell83].)

The communications level also contains a broadcast operation to permit Levels 10, 11, and 12 to request mapping information from their counterparts on other machines. For example, if the file level on one machine cannot locally open the file named by a given capability, it can broadcast that capability to the file levels of other machines; the machine actually holding the file responds with enough information to allow the broadcaster to complete its pending open operation.

5. FILES (Level 10)

Level 10 implements a long-term store for *files*. A file is a named string of bits of known, but arbitrary length and is potentially accessible from all machines in the network. The operations for files are summarized in Table 5.

To establish a connection with a file, a process must present a file capability to the OPEN_FILE operation, which will find the file in secondary storage and allocate buffers for transmissions between the file and the caller. The transmissions themselves are requested by READ_FILE and WRITE_FILE operations. Each read operation copies a segment of information from the file to the caller's virtual memory and advances a read pointer by the length of the segment. Each write operation appends a segment from the caller's virtual

TABLE 5: Specification of Files Level Interface (Level 10).

Form of call	Effect
file_cap := CREATE_FILE( )	Creates a new empty file and returns a capability for it. (If the caller is a user process, it can store this capability in a directory entry and make the file available throughout the system.)
DESTROY_FILE(file_cap)	Destroy the given file (undo a create-file operation).
op_file_cap := OPEN_FILE(file_cap, rw)	Opens the file named by the file capability by allocating storage for buffers and setting up a descriptor block. The value of rw (read, write, or both) is put in the access field of the open-file capability. The read pointer is set to zero and the write pointer to the file's length. (Fails if the file is already open.)
READ_FILE WRITE_FILE CLOSE_FILE	These have the same effects as the READ, WRITE, and CLOSE operations described in the section on stream I/O.
REWIND(op_file_cap)	Reset read pointer to zero.
ERASE(op_file_cap)	Set file length and write pointer to zero; release secondary storage blocks occupied by the file.

memory to the end of the file.

In a multi-machine system, the file level must deal with the problem of nonlocal files. What happens when a process on one machine requests to open a file stored on another machine? There are two alternatives:

1. Open a pair of pipes to Level 10 on the file's home machine; read and write requests are relayed via the forward pipe for remote execution; results are passed back over the reverse pipe. (This is called remote open.)
2. Move the file from its current machine to the machine on which the file is being opened; thereafter all read and write operations are local. (This is called file migration.)

Both methods are feasible. An instance of remote open is in the Berkeley COCANET system [Rowe82]. An instance of migration is in the Purdue STORK file system [Pari83].

The open connection descriptor block for a file, which is addressed by an open-file capability, indicates whether read and write operations can be performed locally or must interact with a surrogate process on another machine. In the latter case, the required open-pipe capability will be implanted in the descriptor block by the open-file command.

Figure 3 illustrates the types of capabilities generated and used during a typical file-editing session.

One important improvement to the basic file system is to allow multiple readers and writers by building in to the read and write operations a solution to the "readers and writers" synchronization problem [Holt78]. Another is to use a version control system to automatically retain different revisions of a file; the file system can then provide access to the older versions when needed [Tich82].

6. DEVICES (Level 11)

The devices level implements a common interface to a wide range of external input and output devices -- for example, terminal displays and keyboards, printers, plotters, time-of-day clock, and optical readers. The interface attempts to hide differences among devices by making input devices appear as sources of data streams and output devices appear as sinks. Obviously, the differences cannot be completely hidden -- for example, cursor-positioning commands must be embedded in the data stream sent to a graphics display -- but a surprising amount of uniformity can be achieved.

Corresponding to each device is a *device driver* program that translates commands at the interface to instructions for operating that device. A considerable amount of effort may be

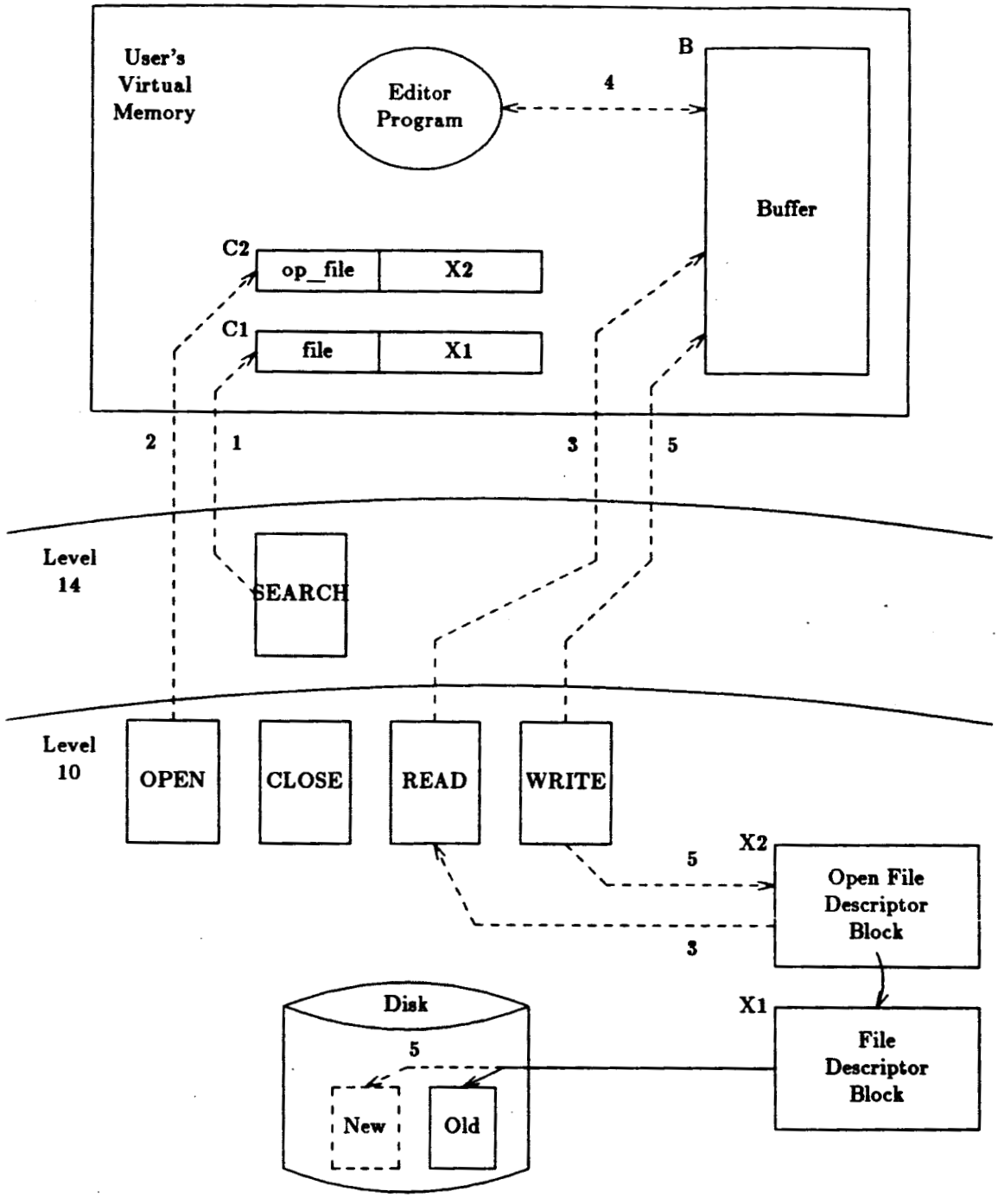

FIGURE 3. The steps of an editing session generate and use various capabilities. 1) Convert the external name string to a capability, *c1*; this can be done by a directory-search command. 2) Open the file for reading and writing by the command *c2* := OPEN_FILE (*c1*, *RW*). 3) Copy the file into a buffer by the command READ_FILE(*c2*, *B*, *all*). 4) Edit the contents of the buffer. 5) Replace the older version of the file by the pair of commands "ERASE(*c2*); WRITE_FILE(*c2*, *B*, *L*)", where *L* is the length of the buffer. 6) Close the file by the command CLOSE_FILE(*c2*).

required to construct a reliable, robust device driver. When a new device is attached to the system, its physical address is stored in a special file accessible to the device drivers.

Table 6 summarizes the interface for external devices.

7. STREAM I/O (Level 12)

An important principle adopted in the hypothetical operating system we are describing here is input-output independence. At Levels 9, 10, and 11 the same fundamental operations (namely OPEN, CLOSE, READ, and WRITE) are defined for pipes, files, and devices. Although writing a block of data to a disk calls for a sequence of events quite different from that needed to supply the same data to the laser printer or to the input of another program, the author of a program does not need to be concerned with those differences. All read and

TABLE 6: Specification of Devices Level Interface (Level 11).

Form of call	Effect
dev_cap := CREATE_DEV(type, address)	Returns a capability for a device of the given *type* at the given *address*. The access code of the returned capability will not include "W" if the device is read-only or "R" if the device is write-only.
DESTROY_DEV(dev_cap)	Detach the given device from the system (undo a create-device operation).
op_dev_cap := OPEN_DEV(dev_cap, rw)	Opens the device named by the device capability by allocating storage for buffers and setting up a descriptor block. The value in the access field of the open-device capability is the logical **AND** of *rw* and the access code of the device capability. (Fails if the device is already open.)
READ_DEV WRITE_DEV CLOSE_DEV	These have the same effects as the READ, WRITE, and CLOSE operations described in the section on stream I/O.

write statements in a program can refer to input and output *ports*, which are attached to particular files, pipes, or devices only when the program is executed.

This strategy, which is an instance of delayed binding, can greatly increase the versatility of a program. A library program (such as the pattern-finding "grep" program in UNIX) can take its input from a file or directly from a terminal and can send its output to another file, to a terminal, or to a printer. Without delayed binding, each program would have to be written to handle each possible combination of source and destination.

A common model of data must be used for pipes, files, and devices. The simplest possibility is the *stream model* in which these objects are media for holding streams of bits. Corresponding to each of these objects is a pair of pointers, R for reading and W for writing; R counts the number of bytes read thus far and similarly for W. Each read request begins at position R and advances R by the number of bytes read. Similarly, each write request begins at position W and advances W by the number of bytes written.

The blocks of data moved by read or write requests are called segments; *seg(x,n)* denotes a contiguous sequence of n bytes beginning at position x in a given data stream. The exact interpretation of a read (or write) request depends on whether the segment comes from a pipe, file, or device. For example, a read request can only be applied at the output end of a pipe and the reader is required to wait until the writer has supplied enough data to fill the request. An output-only device, such as a laser printer, cannot be read and an input-only device, such as a terminal keyboard, cannot be written.

The OPEN operation of Level 12 returns an op_T_cap, corresponding to a given T_cap for T, a pipe, file, or device. The op_T_cap represents an active connection through which data may be passed efficiently to and from the object. The READ and WRITE operations request segment moves across such a connection. The CLOSE operation breaks the connection.

Because the stream model has already been incorporated into the pipes, files, and devices levels, the only new mechanism is a way of switching from a Level-12 operation to its counterpart in the level for the type of object connected to a port. For example, OPEN(*T_cap*, rw) means

```
CASE T OF
    pipe: RETURN OPEN_PIPE(T_cap, rw);
    file: RETURN OPEN_FILE(T_cap, rw);
    dev:  RETURN OPEN_DEV(T_cap, rw);
    ELSE: error;
END CASE
```

Table 7 summarizes the interpretations of the four operations of OPEN, CLOSE, READ, and WRITE for the three kinds of input-output object.

The stream model is not used in every operating system. For example, in Multics, segments are explicit components of the virtual memory; there is no need for a separate concept of file because segments are retained indefinitely until deleted by their owners [Orga72]. In Multics the four operations of Table 7 are implicit. The first time a process refers to a segment, a "missing-binding" interrupt causes the operating system to load and bind that segment to the process. The process can thereafter read or write the segment using the ordinary virtual-addressing mechanism. Certain segments of the address space are permanently bound to devices; reading or writing those segments is equivalent to reading or writing the device. There is no concept of pipe, but the interprocess communication mechanism allows a data stream to be transmitted from one process to another.

8. USER PROCESSES (Level 13)

A user process is a virtual machine containing a program in execution. It consists of a primitive process, a virtual memory, a list of arguments passed as parameters, a list of ports,

TABLE 7: Semantics of I/O Operations on Objects (Level 12)

Form of call	Pipe	File	Device
op_T_cap := OPEN(T_cap, rw)	Verify that T_cap refers to an unopened pipe. Use OPEN_PIPE(T_cap, rw) to initialize an open-pipe descriptor block in which $R = W = 0$; return the op_pipe_cap.	Verify that T_cap refers to an unopened file. Use OPEN_FILE(T_cap, rw) to initialize an open-file descriptor block in which $R = 0$ and $W = L$ (file length); return the op_file_cap.	Verify that T_cap refers to an unopened device. Use OPEN_DEV(T_cap, rw) to initialize an open-device descriptor block in which $R = 0$ or $W = 0$, according as the device is input or output; return the op_dev_cap.
READ(op_T_cap, a, n)	Wait until $R + n \leq W$. Invoke READ_PIPE(op_pipe_cap, a, n) to copy *seg* (R, n) to *seg* (a, n) and advance R to $R + n$. If n=*all*, return immediately with whatever is in the pipe, *seg* $(R, W - R)$.	Set $m = \min\lvert L - R, n \rvert$. Invoke READ_FILE(op_file_cap, a, m) to copy *seg* (R, m) to *seg* (a, m). If n=*all*, return immediately with whatever is in the file, *seg* $(R, W - R)$.	Invoke READ_DEV(op_dev_cap, a, m) as for file. If n=*all*, return immediately with whatever input is available, *seg* $(R, W - R)$. (No effect for output device.)
WRITE(op_T_cap, a, n)	Invoke WRITE_PIPE(op_pipe_cap, a, n) to copy *seg* (a, n) to *seg* (W, n) and advance W to $W + n$. (May awaken waiting reader).	Invoke WRITE_FILE(op_file_cap, a, n) as for pipe, plus advance L to $L + n$.	Invoke WRITE_DEV(op_dev_cap, a, n) as for pipe. (No effect for input device.)
CLOSE(op_T_cap)	If the pipe contains a waiting reader, return to that reader the remaining segment in the pipe. Invoke CLOSE_PIPE(op_pipe_cap) to deallocate the open-pipe descriptor block.	Invoke CLOSE_FILE(op_file_cap) to deallocate the open-file descriptor block.	Invoke CLOSE_DEV(op_dev_cap) to deallocate the open-device descriptor block.

and context. Each "port" is a capability for an open pipe, file, or device. The "context" is a set of variables characterizing the environment in which the process operates; it includes the current working directory, the command directory, a link to the parent process, a linked list of spawned processes, and a signal variable that counts the number of spawned processes whose execution is yet incomplete. Figure 4 illustrates the format of a user process descriptor block.

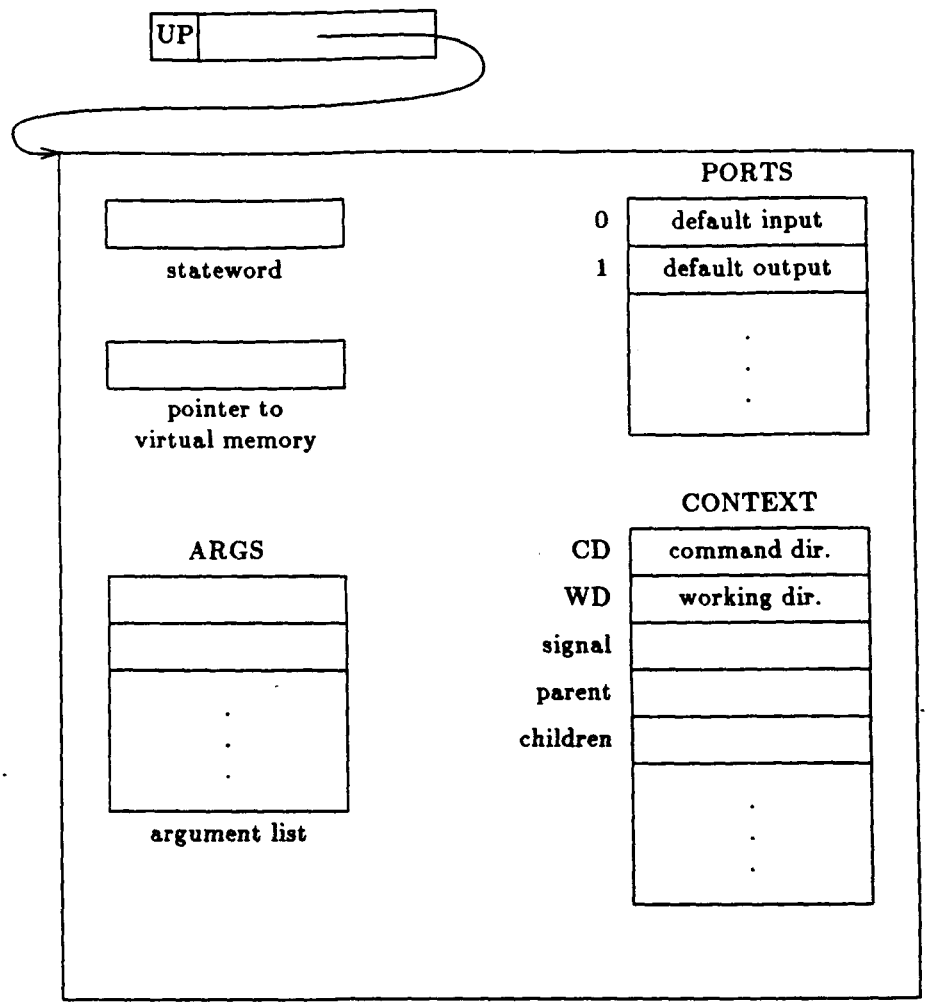

FIGURE 4. A user process is a virtual machine created for the purpose of executing a given program. It contains a primitive process, a virtual memory holding the given program, a list of arguments supplied at the time of call, a list of ports, and a set of context variables. By convention, PORTS[0] is the default input and PORTS[1] is the default output; these two ports are bound to pipes, files, or devices when the process is created. The process can open other ports as well after it commences execution.

A new user process is created by a FORK operation. The creator is called the "parent" and the new process a "child". A parent can exercise control over its children by resuming, suspending, or killing them. A parent can stop and wait for its children to complete their tasks by a join operation, and a child can signal its completion by an exit operation. Table 8 summarizes.

TABLE 8: Specification of User Process Operations (Level 13)

Form of Call	Effect
up_cap := FORK(file_cap, params, in, out)	Allocate a user process descriptor block. Create a suspended primitive process and store its index in a new user-process capability. Create a virtual memory and load the executable file denoted by file_cap. Copy the parameters into the ARGS list. Verify that *in* and *out* are capabilities for pipes, files, or devices; if so, open *in* for reading and put the open-capability in PORTS[0], and open *out* for writing and put the open-capability in PORTS[1].
JOIN(A)	Wait until caller's context variable, *signal*, is *A*, then return.
KILL(up_cap)	Terminate the designated user process, but only if it is a child of the caller. This entails destroying the primitive process and virtual memory, closing open pipes, files, or devices connected to ports, releasing the storage held by the descriptor block, and removing the deleted process from the list of the caller's children.
EXIT	Terminate the caller process and add 1 to the *signal* variable of the parent process.
SUSPEND(up_cap)	Put the primitive process contained within the user process "up_cap" into the suspended state, but only if the caller is the parent of process "up_cap".
RESUME(up_cap)	Put the primitive process contained within the user process "up_cap" into the ready state, but only if the caller is the parent of process "up_cap".
op_T_cap := OPEN(T_cap,rw)	Invoke the OPEN command in Level 12, store a copy of the result in the next available position in the PORTS table and return the result to the caller (T is pipe, file, or device0.

Notice that an OPEN operation appears in Table 8. This OPEN operation hides the Level-12 OPEN from higher levels. It allows Level 13 to store copies of all open-object capabilities in the PORTS table. When a process terminates, Level 13 can assure that all open objects are closed by invoking the Level-12 CLOSE operation for each entry in the PORTS table.

9. DIRECTORIES (Level 14)

Level 14 is responsible for managing a hierarchy of directories containing capabilities for sharable objects. In our hypothetical system, these are pipes, files, devices, directories, and user processes; capabilities for open pipes, files, and devices are not sharable and cannot appear in directories. A hierarchy arises because a directory can contain capabilities for subordinate directories.

A directory is a table that matches an external name, stored as a string of characters, with an access code and a capability. In a tree of directories, the concatenated sequence of external names from the root to a given object serves as a unique, system-wide external name for that object. A directory system of this kind has been implemented on the Cambridge CAP

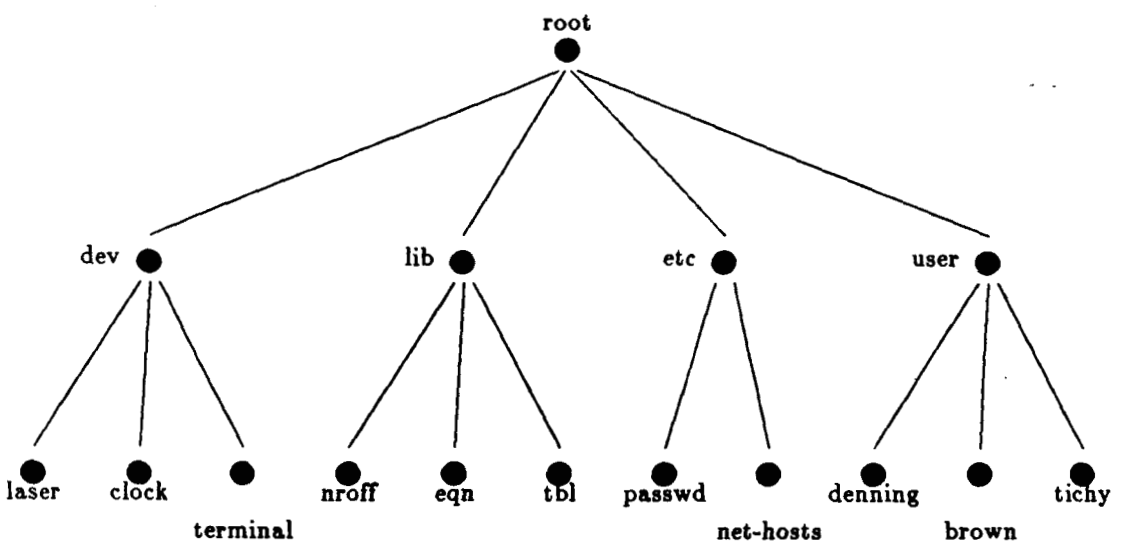

FIGURE 5. A directory hierarchy can be depicted as an inverted tree whose topmost node is called "root". Some directories are permanently reserved for specific purposes. For example, the *dev* directory lists all the external devices of the system. The *lib* directory lists the library of all the executable programs maintained by the system's administration. A *user* directory contains a subdirectory for each authorized user; that subdirectory is the root of a subtree belonging to that user. In UNIX, the unique external name of an object is formed by concatenating the external names along the path from the root, separated by slash (/) and omitting "root". Thus the laser printer's external name is "/dev/laser".

machine [Wilk79].

The principal operation of Level 14 is a search command that locates and returns the capability corresponding to a given external name. Thus the directory level is merely a mechanism for mapping external to internal names. Only one type of capability can be mapped to an object at this level: a directory capability. All other capabilities must be presented to their respective levels for interpretation. Information about object attributes, such as ownership or time of last use, is not kept in directories; it is kept in the object descriptor blocks within the object manager levels.

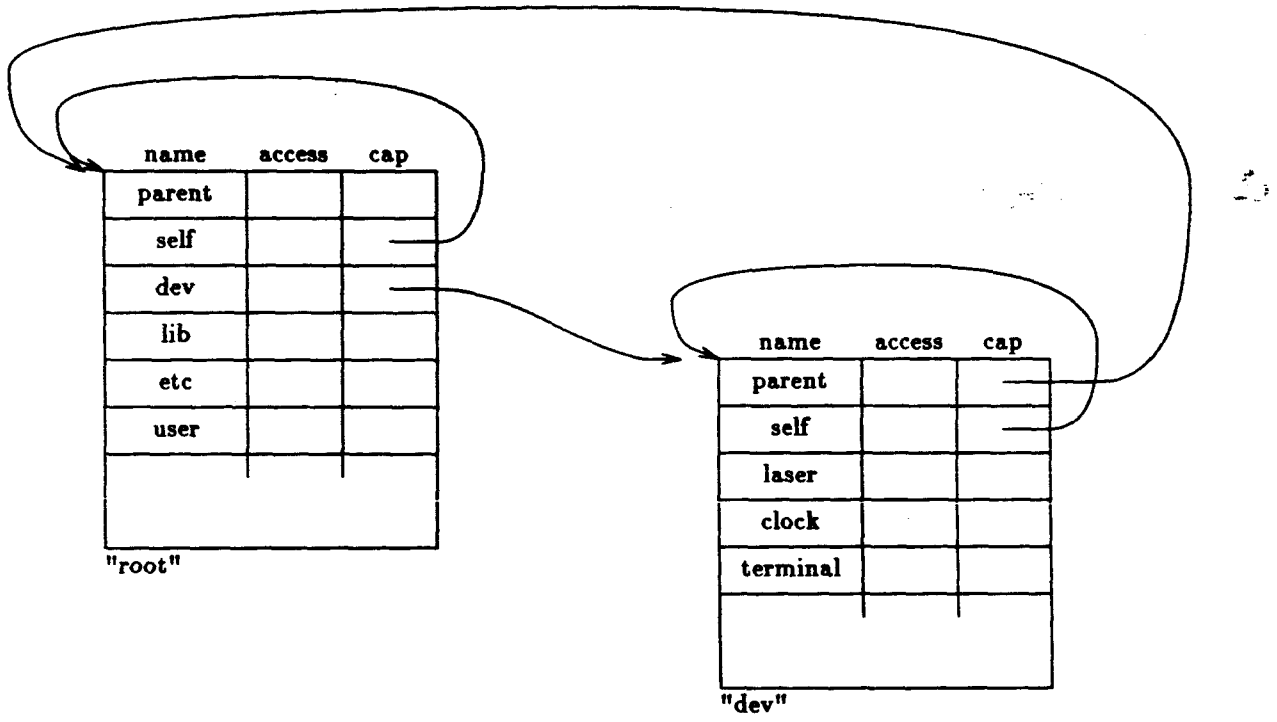

FIGURE 6. A directory is a table matching an external name string with an access code and a capability. Every directory contains a capability pointing to its immediate parent and a capability pointing to itself; the self-capability is can be used to fill in the parent entry in a new subordinate directory. Because directories are at a higher level than files, the file system can be used to store directories. A directory containing only the self and parent entries is considered empty.

The requirement for system-wide unique names implies that the directory level also has the responsibility for ensuring that portions of the directory hierarchy resident on each machine are consistent. This can be accomplished by methods for replication in a distributed database system [Seli80]. To control the number of update messages in a large system, the full directory database may be kept on only a small subset of machines (e.g., two or three) implementing a *stable store*. Copies of the views of the directory database being accessed by a given user can be stored in a workstation or other local system after that user logs in. Operations that modify an entry in a directory must send updates to the stable-store machines, which relay them to affected workstations.

Specifications of the principal operations of the directory level are given in Table 9. These operations allow higher-level programs to create objects and store capabilities for them in directories. This table is not a complete specification of a directory manager; for example, it contains no command to change the name and access fields of a directory entry.

The attach operation is used to create new new entry in a directory. The access field of the capability returned by a search operation will be the conjunction of the entry's access code and the access field already in the capability.

In the special case of attaching a directory to a directory, the attach operation must also define the parent of the newly attached directory. The operation fails if a parent is already defined (see Figure 6). The detach operation only removes entries from directories but has no effect on the object to which a capability points. To destroy an object, the destroy operation of the appropriate level must be used. To minimize inadvertent deletions, the destroy-directory operation fails if applied to a nonempty directory.

The attach and detach operations must notify the stable store so that changes become effective throughout the system. To keep this simple, we have required a) that an empty

TABLE 9: Specification of a Directory Manager Interface (Level 14).

Form of Call	Effect
dir_cap := CREATE_DIR(access)	Allocate an empty directory. Return a capability with its permission bits set to the given access code. (This directory is not attached to the directory tree.)
DESTROY_DIR(dir_cap)	Destroy (remove) the given directory. (Fails if the directory is nonempty.)
ATTACH(obj_cap, dir_cap, name, access)	Make an entry called *name* in the given directory (*dir_cap*); store in it the given object-capability (*obj_cap*) and the given access code. If *obj_cap* denotes a directory, set its parent entry from the self entry of the directory *dir_cap*. Notify the directory stable store of the change. (Fails if the name already exists in the directory *dir_cap*, if the directory *dir_cap* is not attached, or if *obj_cap* denotes an already-attached directory.)
DETACH(dir_cap, name)	Remove the entry of the given *name* from the given directory. Notify the directory stable store of the change. (Fails if the name does not exist in the given directory or if the given directory is nonempty.)
obj_cap := SEARCH(dir_cap, name)	Find the entry of the given *name* in the given directory and return a copy of the associated capability. Set the access field in the returned capability to the minimum privilege enabled by the access fields of the directory entry and of the capability. (Fails if the name does not exist in the given directory.)
seg := LIST(dir_cap)	In a segment of the caller's virtual memory, return a copy of the contents of the directory. (A user-level program can interrogate the other levels for other information about the objects listed in the directory -- e.g., date of last change.)

directory must first be attached to the global directory tree before entries are made in it, and b) that a directory must be empty before being detached. A more complicated notification mechanism will be needed if a process is allowed to construct a directory subtree before attaching its root to the global directory tree.

10. SHELL (Level 15)

Most users of the system spend most of their time employing existing programs, not writing new ones. When a user logs in, the operating system creates a user process containing a copy of the shell program with its default input connected to the user's keyboard and its default output connected to the user's display. The shell is the program that listens to the user's terminal and interprets the input as commands to invoke existing programs in specified combinations and with specified inputs.

The shell scans each complete line of input to pick out the names of programs to be invoked and the values of arguments to be passed to them. For each program called in this way, the shell creates a user process. The user processes are connected according to the data flow specified in the command line.

Operations of substantial complexity can be programmed in the command language of the UNIX shell. For example, the operations that format then print a file named "text" can be set in motion by the command line:

tbl < text | eqn | lptroff > output

The first program is *tbl*, which scans the data on its input stream and replaces descriptions of tables of information with the necessary formatting commands. The "<" symbol indicates that *tbl* is to take its input from the file "text". The output of *tbl* is directed by a pipe (the "|" symbol) to the input of *eqn*, which replaces descriptions of equations with the necessary formatting commands. The output of *eqn* is then piped to *lptroff*, which generates the commands for the laser printer. Finally, the ">" symbol indicates that the output of *lptroff* is to be placed in a file named "output". If "> output" were replaced with "| laser", the data would instead be sent directly to the laser printer.

Having identified the components of a command line, the shell obtains capabilities for them by a series of commands:

```
c1 := SEARCH(CD, "tbl");
c2 := SEARCH(WD, "text");
c3 := CREATE_PIPE();
c4 := SEARCH(CD, "eqn");
c5 := CREATE_PIPE();
c6 := SEARCH(CD, "lptroff");
c7 := CREATE_FILE();
ATTACH(c7, WD, "output", all);
```

The variable "CD" holds a capability for a commands directory and "WD" holds a capability for the current working directory. Both CD and WD are part of the shell's context (see Figure 4).

The shell then creates and resumes user processes that execute the three components of the pipeline and awaits their completion:

```
RESUME( FORK(c1, -, c2, c3) );
RESUME( FORK(c4, -, c3, c5) );
RESUME( FORK(c6, -, c5, c7) );
JOIN(3);
```

After the JOIN returns, the shell can kill these processes and acknowledge completion of the entire command to the user (by a "prompt" character).

If the specification "< text" were omitted, the shell would have connected *tbl* to the default input, which is the same as its own, namely the terminal keyboard. In this case, the second search command would be omitted and the first fork operation would be

```
FORK(c1, -, PORTS[0], c3)
```

Similarly, if "> output" were omitted, the shell would have connected *lptroff* to the default output, the shell's PORTS[1].

If an elaborate command line is to be performed often, typing it can become tedious. UNIX encourages users to store complicated commands in executable files called *shell-scripts* that become simpler commands. A file named *lp* might be created with the contents:

```
tbl < $1 | eqn | lptroff > $2
```

where the names of input and output files have been replaced by variables $1 and $2. When the command *lp* is invoked, the variables $1 and $2 are replaced by the arguments following the command. For example, typing

```
lp text output
```

would substitute "text" for $1 and "output" for $2 and so would have exactly the same effect as the original command line.

11. INITIALIZATION

One small but essential piece of an operating system has not been discussed -- the method of starting up the system. The startup procedure, called a *bootstrap sequence*, begins with a very short program copied into the low end of main memory from a permanent read-only memory. This program loads a longer program from the disk, which then takes control and loads the operating system itself. Finally, the operating system creates a special login process connected to each terminal of the system.

When a user correctly types an identifier and a password, the login process will create a shell process connected to the same terminal. When the user types a logout command, the shell process will exit and the login process will resume its vigil over the terminal.

12. CONCLUSION

We have used the levels model to describe the functions of contemporary multi-machine operating systems. This description shows how it is possible to systematically hide the physical locations of all sharable objects and yet be able to locate them quickly when given a name in the directory hierarchy.

The directory function can be generalized from its traditional role by storing capabilities, rather than file identifiers, in directory entries. No user machine need have a full, local copy of the directory structure; it need only encache the view with which it is currently working. The full structure is maintained by a small group of machines implementing a stable store.

The model can deal with heterogeneous systems consisting of general purpose user machines, such as workstations, and special purpose machines, such as stable stores, file servers, and supercomputers. Only the user machines need contain a full operating system; the special purpose machines require only a simple operating system capable of managing local tasks and communicating on the network.

The levels model is based on the same principle found in nature to organize many scales of space and time. At each level of abstraction there are well defined rules of interaction for the objects visible at that level; the rules can be understood without detailed knowledge of the smaller elements making up those objects. The many parts of an operating system cannot be fully understood without recourse to this principle.

13. ACKNOWLEDGEMENTS

We are grateful to many colleagues for advice and counsel while we formulated and refined this model. These include J. Dennis, E. Dijkstra, N. Habermann, B.Randell, and M. Wilkes for early inspirations about hierarchical structure; K. Levitt, M. Meliar-Smith, and P.

Neumann for numerous discussions about PSOS; D. Cheriton, for demonstrating in the THOTH system, D. Comer, for demonstrating in the XINU system, and G. Popek, for demonstrating in the LOCUS system, many of the ideas of this paper; D. Schrader for advice on distributed databases and help with the document; D.Denning for advice on access controls, capability systems, and verification; A. Birrell, B. Lampson, R. Needham, and M. Schroeder for advice on "server models" of distributed systems; and D. Farber, A. Hearn, T. Korb, and L. Landweber for advice on networks through the CSNET Project. We are grateful to the National Aeronautics and Space Administration, which supported part of this work under contract NAS2-11530 at RIACS. We are also grateful to the National Science Foundation, which supported part of this work through grant MCS-8109513 at Purdue University.

14. REFERENCES

Bell83.
Bell Laboratories,, "UNIX, System User's Manual: System V," 301-905 Issue 1, Western Electric (January 1983).

Birr82.
Birrell, Andrew D., Roy Levin, Roger M. Needham, and Michael D. Schroder, "Grapevine: An Exercise in Distributed Computing," *Communications of the ACM* **25**(4) pp. 260-274 (April 1982).

Bogg83.
Boggs, David R., "Internet Broadcasting," CSL-83-3, XEROX PARC, Palo Alto (October 1983).

Brin73.
Brinch Hansen, P., *Operating System Principles,* Prentice-Hall, Englewood Cliffs, NJ (1973).

Cher82.
Cheriton, David R., *The Thoth System: Multi-process Structuring and Portability,* Elsevier Science, New York (1982).

Cher84.
Cheriton, David R., "The V Kernel: A Software Base for Distributed Systems," *IEEE Software* **1**(2) pp. 19-42 (April 1984).

Come84.
Comer, Douglas, *Operating System Design: The XINU Approach,* Prentice-Hall, Englewood Cliffs (1984).

Denn70.
Denning, Peter J., "Virtual Memory," *Computing Surveys* **2**(3) pp. 154-216 (September

1970).

Denn71.
Denning, Peter J., "Third Generation Computer Systems," *Computing Surveys* **3**(4) pp. 175-212 (December 1971).

Denn76.
Denning, Peter J., "Fault-Tolerant Operating Systems," *Computing Surveys* **8**(4) pp. 359-389 (December 1976).

Denn81.
Denning, Peter J., T. Don Dennis, and Jeffrey A. Brumfield, "Low Contention Semaphores and Ready Lists," *Communications of the ACM* **24**(10) pp. 687-699 (October 1981).

Denn66.
Dennis, J. B. and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM* **9**(3) pp. 143-155 (March 1966).

Dijk68.
Dijkstra, Edsger W., "The Structure of the THE-Multiprogramming System," *Communications of the ACM* **11**(5) pp. 341-346 (May 1968).

Fabr74.
Fabry, R. S., "Capability-Based Addressing," *Communications of the ACM* **17**(7) pp. 403-412 (July 1974).

Gold74.
Goldberg, Robert P., "Survey of Virtual Machine Research," *Computer*, pp. 34-46 (June 1974).

Habe76.
Habermann, A. Nico, Lawrence Flon, and Lee W. Cooprider, "Modularization and Hierarchy in a Family of Operating Systems," *Communications of the ACM* **19**(5) pp. 266-272 (May 1976).

Holt78.
Holt, R. C., E. D. Lazowska, G. S. Graham, and M. A. Scott, *Structured Concurrent Programming with Operating Systems Applications*, Addison-Wesley, Reading, MA (1978).

IBM73.
IBM,, "IBM Virtual Machine Facility/370: Introduction," GC20-1800-1, IBM (August 1973).

Jone79.
Jones, Anita K., Robert J. Chansler Jr., Ivor Durham, Karsten Schwans, and Steven R. Vegdahl, "StarOS, A Multiprocessor Operating System for the Support of Task Forces," *Proceedings of the Seventh Symposium on Operating Systems Principles*, pp. 117-127 (December 1979).

Kern84.
Kernighan, Brian W. and Rob Pike, *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs (1984).

Levy84.
Levy, Henry M., *Capability-Based Computer Systems*, Digital Press, Bedford, MA (1984).

Neum80.
Neumann, Peter G., Robert S. Boyer, Richard J. Feiertag, Karl N. Levitt, and Lawrence Robinson, "A Provably Secure Operating System, its Applications, and Proofs," CSL-116 (2nd edition), SRI International, Menlo Park, CA (May 7, 1980).

Orga83.
Organick, Elliott, *A Programmer's View of the Intel 432 System,* McGraw-Hill, New York (1983).

Orga72.
Organick, Elliot I., *The Multics System: An Examination of its Structure,* The MIT Press, Cambridge, MA (1972).

Oust80.
Ousterhout, John K., Donald A. Scelza, and Pradeep S. Sindhu, "Medusa: An Experiment in Distributed Operating System Structure," *Communications of the ACM* **23**(2) pp. 92-105 (February 1980).

Pari83.
Paris, Jehan-Francois and Walter F. Tichy, "Stork: An Experimental Migrating File System for Computer Networks," CSD-TR-411, Purdue University, West Lafayette, IN (February 1983).

Pope81.
Popek, G., B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "LOCUS: A Network Transparent, High Reliability Distributed System," *Proceedings of the Eighth Symposium on Operating Systems Principles,* pp. 169-177 (December 1981).

Ritc74.
Ritchie, D. M. and K. L. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM* **17**(7) pp. 365-375 (July 1974).

Rowe82.
Rowe, Lawrence A. and Kenneth P. Birman, "A Local Network Based on the UNIX Operating System," *IEEE Transactions on Software Engineering* **SE-8**(2) pp. 137-146 (March 1982).

Seli80.
Selinger, P. G., "Replicated Data," pp. 223-231 in *Distributed Data Bases*, ed. F. Poole,Cambridge University Press, Cambridge, England(1980).

Tane81.
Tanenbaum, Andrew S., *Computer Networks,* Prentice-Hall, Englewood Cliffs, NJ (1981).

Tich82.
Tichy, Walter F., "Design, Implementation, and Evaluation of a Revision Control System," pp. 58-67 in *Proceedings of the 6th International Conference on Software Engineering*, IPS, ACM, IEEE, NBS(September 1982).

Wilk79.
Wilkes, M. V. and R. M. Needham, *The Cambridge CAP Computer and its Operating System,* Elsevier/North-Holland Publishing Co. (1979).

Wulf81.
Wulf, William A., Roy Levin, and Samuel P. Harbison, *HYDRA/C.mmp, An Experimental Computer System,* McGraw-Hill (1981).